Parallel I/O for Applications

Rob Latham

Mathematics and Computer Science

Argonne National Laboratory

Argonne NATIONAL LABORATORY

... for a brighter future

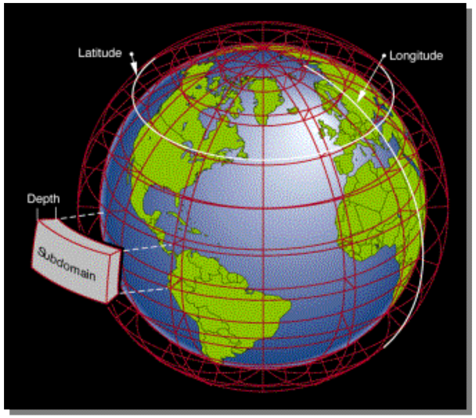
U.S. Department of Energy

THE UNIVERSITY OF CHICAGO

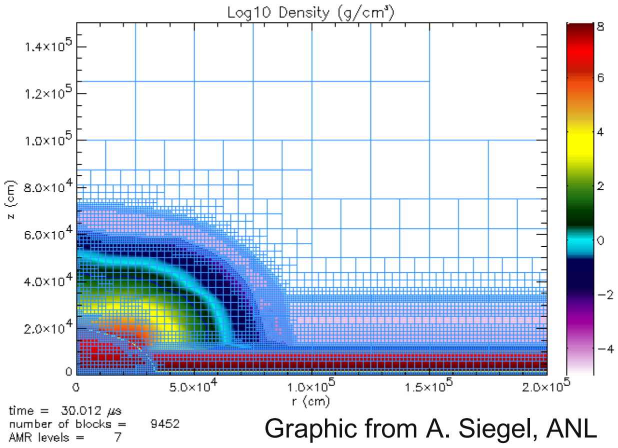
Office of Science
U.S. DEPARTMENT OF ENERGY

Application I/O

- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, variable length records
  - Headers, attributes on data
- I/O systems have very simple data models
  - Tree-based hierarchy of containers
  - Some containers have streams of bytes (files)
  - Others hold collections of other containers (directories or folders)
- Someone has to map from one to the other!



Graphic from J. Tannahill, LLNL



Graphic from A. Siegel, ANL

Common Approaches to Application I/O

- Root performs I/O
  - Pro: trivially simple for “small” I/O
  - Con: bandwidth limited by rate one client can sustain
  - Con: may not have enough memory on root to hold all data
- All processes access their own file
  - Pro: no communication or coordination necessary between processes
  - Pro: avoids some file system quirks (e.g. false sharing)
  - Con: for large process counts, lots of files created
  - Con: data often must be post-processed to recreate canonical dataset
  - Con: uncoordinated I/O from all processes may swamp I/O system
- All processes access one file
  - Pro: only one file (per timestep etc.) to manage: fewer files overall
  - Pro: data can be stored in canonical representation, avoiding post-processing
  - Con: can uncover inefficiencies in file systems (e.g. false sharing)
  - Con: uncoordinated I/O from all processes may swamp I/O system

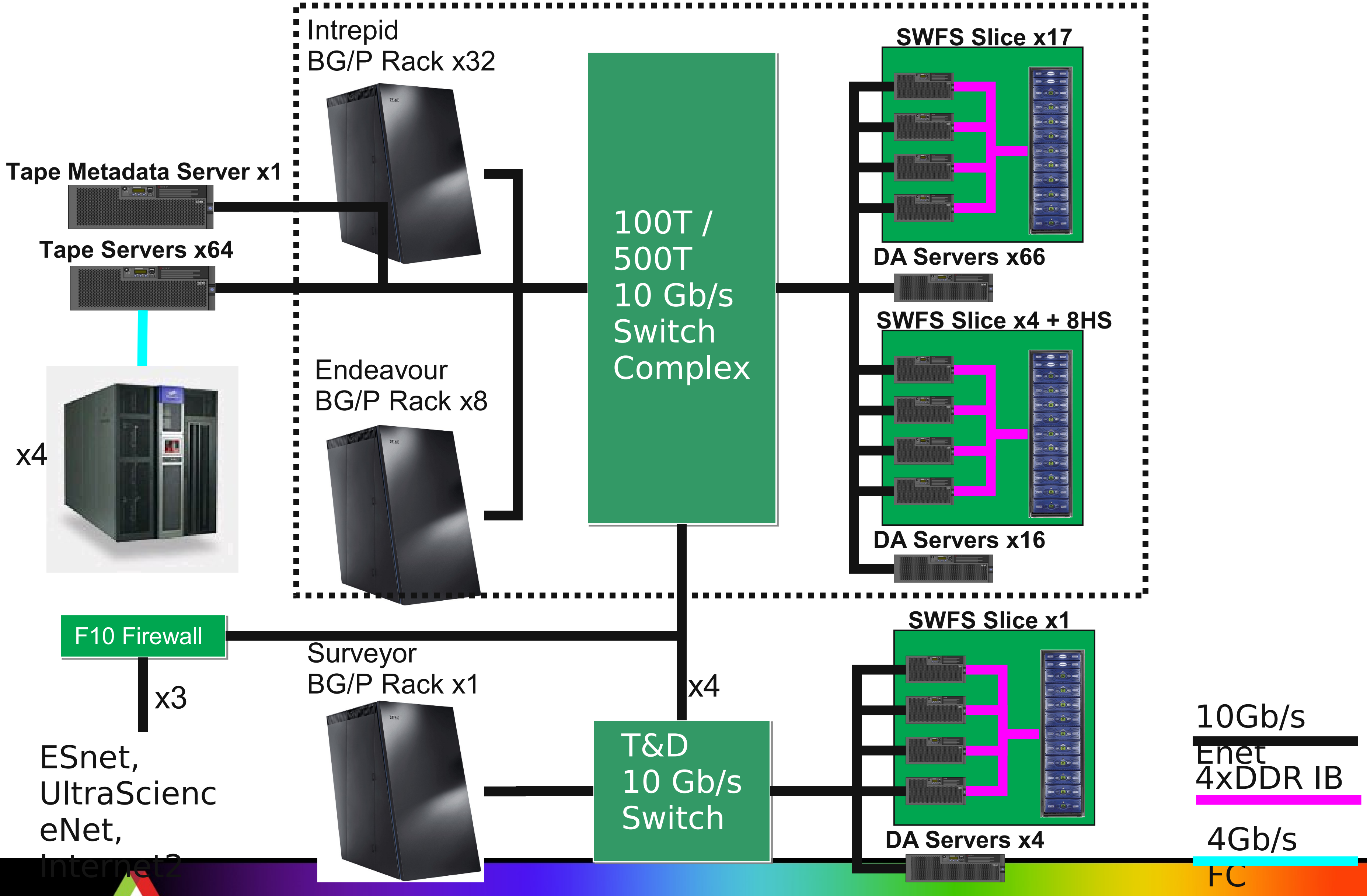
Challenges in Application I/O

- Leveraging aggregate communication and I/O bandwidth of clients
- …But not overwhelming a resource limited I/O system with uncoordinated accesses!
- Limiting number of files that must be managed (also a performance issue)
- Avoiding unnecessary post-processing
- Avoiding file system quirks

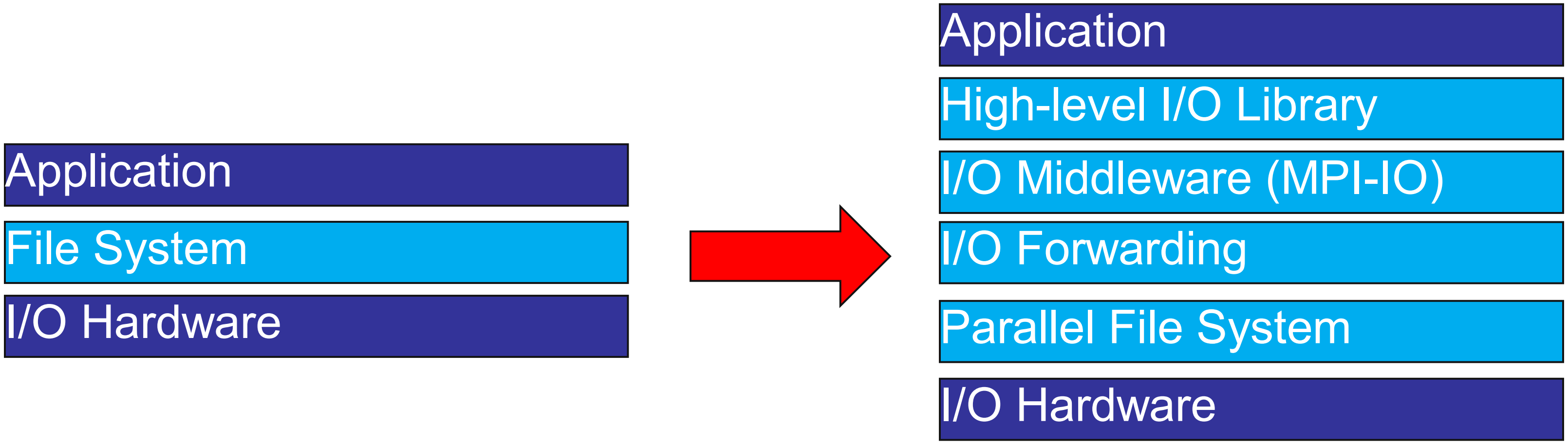
- Often application teams spend so much time on this that they never get any further:
  - Interacting with storage through convenient abstractions
  - Storing in portable formats

- Computer science teams that are experienced in parallel I/O have developed software to tackle all of these problems
  - **Not the application's job.**

Argonne BGP Configuration



Software for Parallel I/O in HPC



- Applications require more software than just a parallel file system
- Support provided via multiple layers with distinct roles:
  - Parallel file system maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)
  - I/O Forwarding found on largest systems to assist with I/O scalability
  - Middleware layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
  - High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
- Goals: scalability, parallelism (high bandwidth), and usability

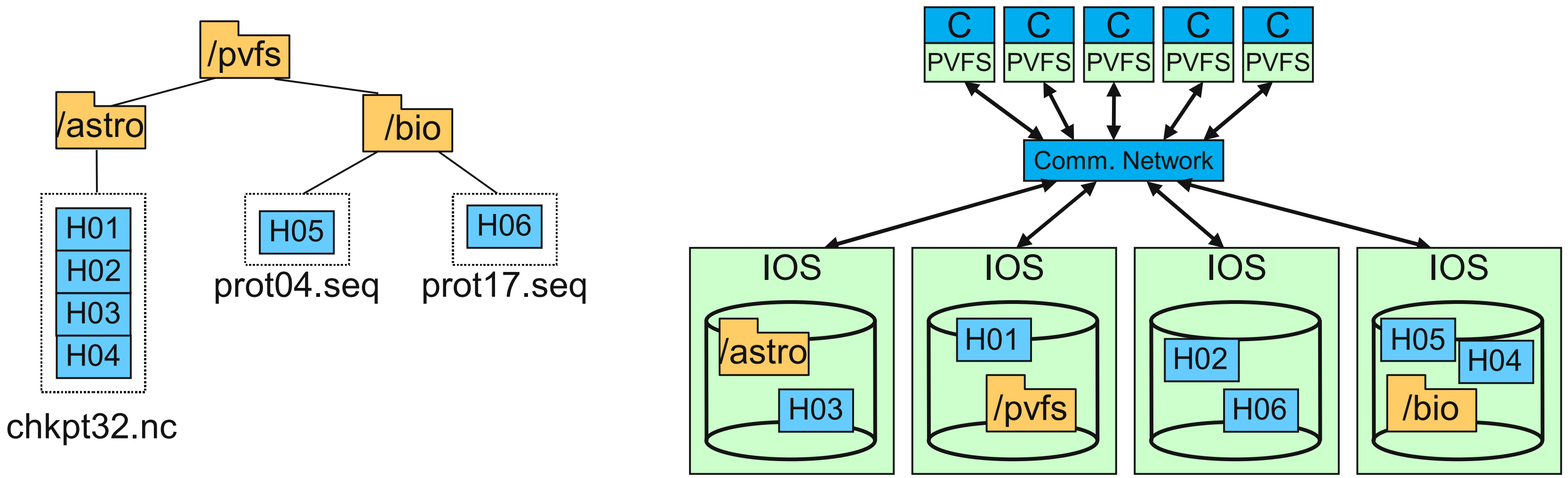
Why All This Software?

"All problems in computer science can be solved by another level of indirection." -- David Wheeler

- Parallel file systems must be general purpose to be viable products
  - Many workloads for parallel file systems still include serial codes
  - Most of our tools still operate on the UNIX "byte stream" file model
- I/O forwarding addresses HW constraints and helps us leverage existing file system implementations at greater (unintended?) scales
- Programming model developers are not (usually) file system experts
  - Implementing programming model optimizations on top of common file system APIs provides flexibility to move to new file systems
  - Again, trying to stay as general purpose as possible
- High level I/O libraries mainly provide convenience functionality on top of existing APIs
  - Specifically attempting to cater to specific data models
  - Enable code sharing between applications with similar models
  - Standardize how contents of files are stored

The Parallel Virtual File System (PVFS)

Parallel Virtual File System (PVFS)

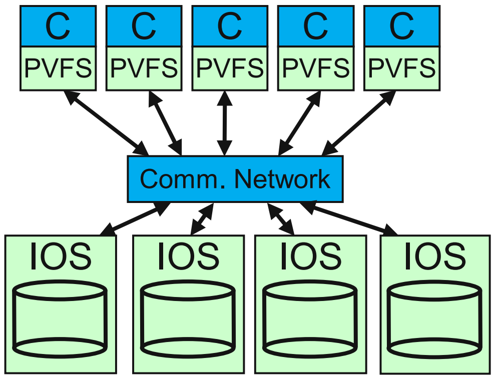


An example PVFS file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS), while small bioinformatics files are each stored on a single IOS.

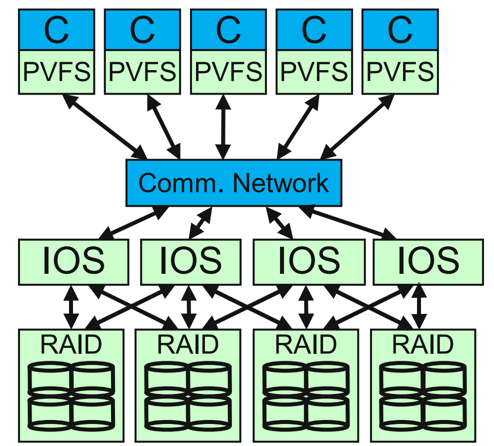
- File-based storage model, very similar to object based storage model
  - Fragments of files stored on distributed IO Servers (IOS)
    - *I/O servers manage their own local storage*
  - Single server type can also store metadata
  - Clients perform accesses in terms of byte ranges in files (region-oriented)
- Available for Linux OS and IBM Blue Gene systems
- Tightly-coupled MPI-IO implementation

PVFS Architecture

- Communication performed over existing cluster network
  - TCP/IP, InfiniBand, Myrinet, Portals
- Servers store data in local file systems (e.g. ext3, XFS)
  - Local files store PVFS file strips
  - Berkeley DB currently used for metadata (rather than files)
- Mixed kernel-space, user-space implementation
  - VFS module in kernel with user-space helper process
  - User-space servers, interface for kernel bypass on clients
- Commodity failover (e.g. Heartbeat) may be used to set up active-active server configuration for both metadata and data

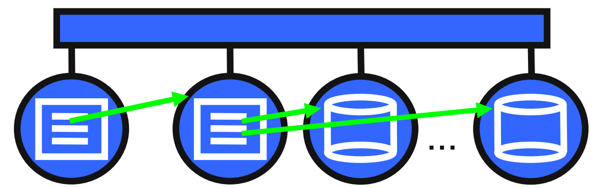


PVFS configured as scratch file system



PVFS configured with redundancy

PVFS Files and Directories



- PVFS files are made up of objects holding data (dataspaces) and a distribution function
- Directory dataspace holds metafile handles
- Metafile dataspace holds
  - Permissions, owner, extended attributes
  - References to dataspaces holding data
  - Parameters for distribution function
- Datafiles hold the file data itself
  - Usually one datafile on each server for parallelism
- Distribution function determines how data in datafiles maps into logical file
  - By default file data is split into 64Kbyte blocks and distributed round-robin into datafiles
  - Because list of datafiles and distribution function don't change, clients may cache this information indefinitely
    - *No communication with server holding metadata during I/O*

State, Consistency, and Caching

- In GPFS and Lustre, clients are allowed to hold on to important file system state
  - Locks are used to keep these in sync with data on storage or to prevent other clients from accessing the data until it is committed
  - Locks (which are state themselves) are further used for atomic I/O
  - Problems: lock traffic is nondeterministic, client death becomes complicated
- PVFS does not hold critical file system state on clients (stateless)
  - Clients may appear and disappear without impacting file system
    - *Much like a web server*
  - PVFS does provide a coherent view of file data
    - *Processes immediately see changes from others*
  - Does not provide atomic writes or reads
    - *Other software responsible for this coordination (e.g. MPI-IO)*
  - Does provide atomic metadata operations
    - *Creating and removing files and directories atomically change the name space*
  - No locks necessary!
- Without locks to maintain coherence, caching possibilities are very limited
  - Clients cache immutable metadata on files allowing I/O without metadata access
  - Data caching restricted to executables and mmapped files (read-only)

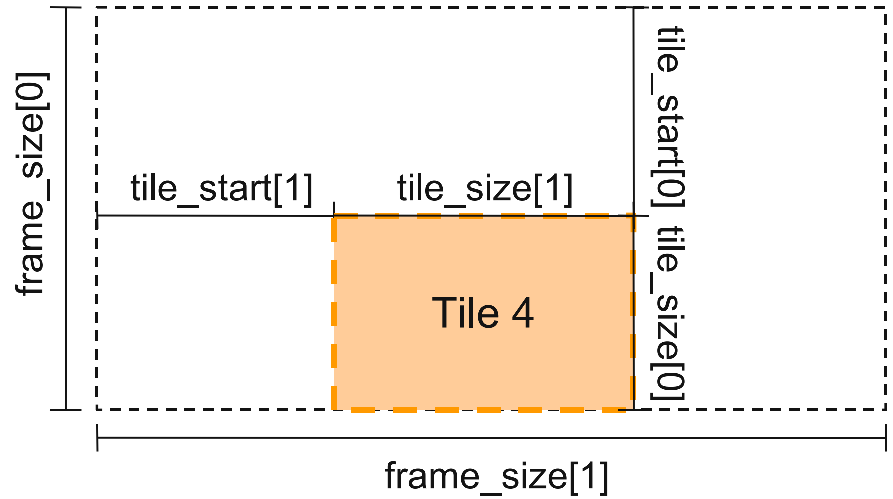
MPI-IO Interface

MPI-IO

- The Message Passing Interface (MPI) is an interface standard for writing message passing programs
  - Most popular programming model on HPC systems
- MPI-IO is an I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)

- Implementations available on most platforms (more later)

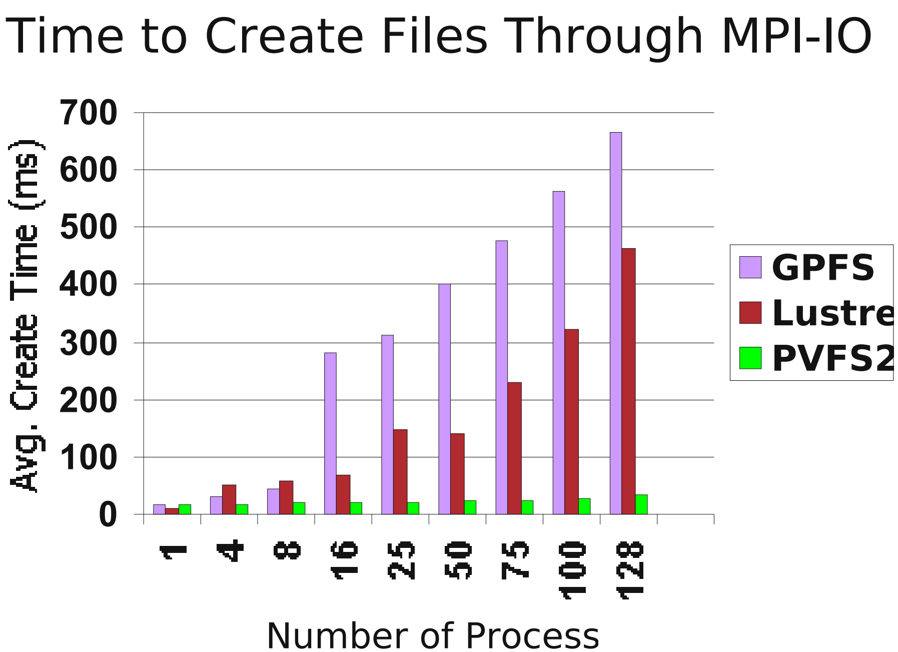
Challenge: Describing Application Data

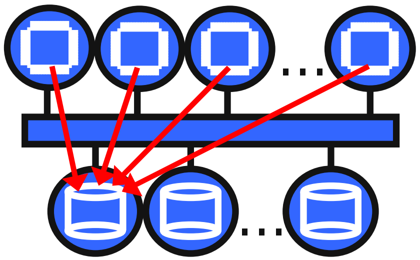


- MPI_Type_create_subarray can describe any N-dimensional subarray of an N-dimensional array
- In this case we use it to pull out a 2-D tile
- Tiles can overlap if we need them to
- Separate MPI_File_set_view call uses this type to select the file region

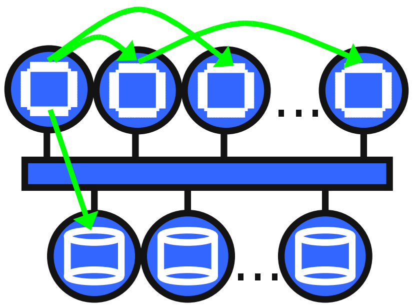
Challenge: Efficient File Creation

- File create rates can actually have a significant performance impact
- Improving the file system interface improves performance for computational science
  - Leverage communication in MPI-IO layer



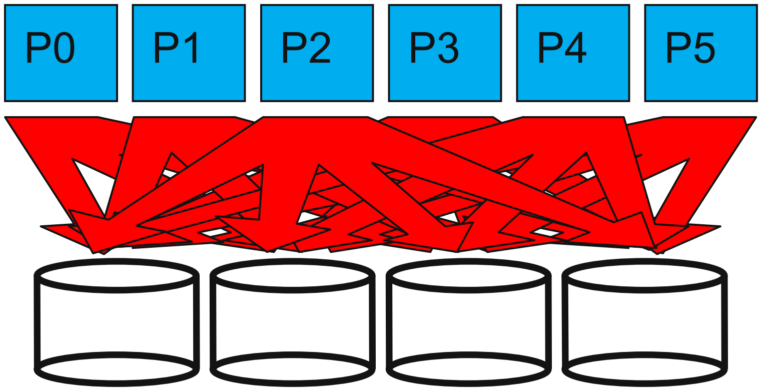


File system interfaces force all processes to open a file, causing a storm of system calls.

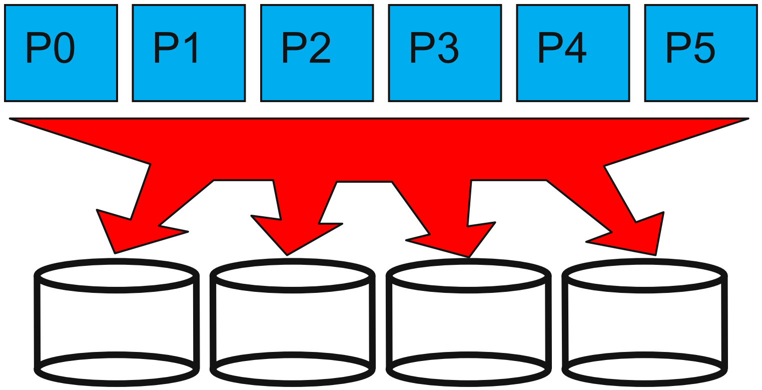


MPI-IO can leverage other interfaces, avoiding this behavior.

Challenge: Coordinating I/O



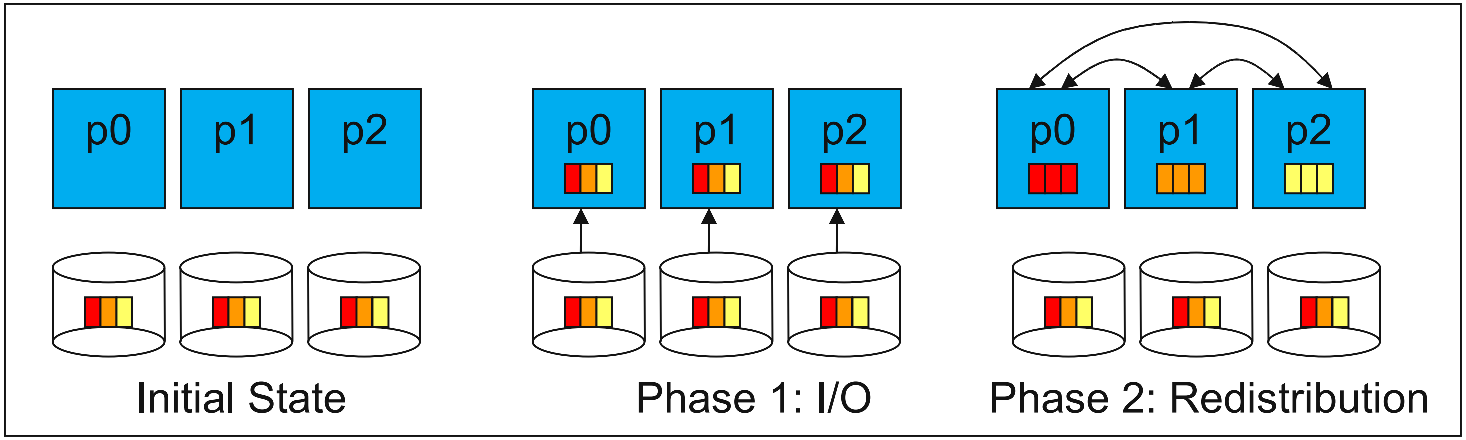
Independent I/O



Collective I/O

- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance

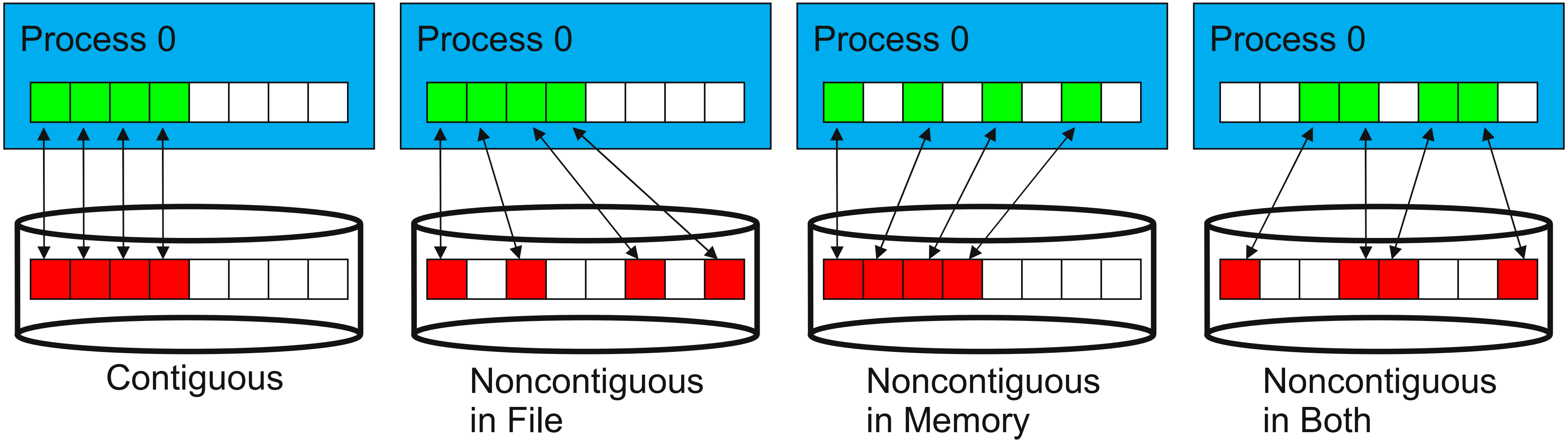
The Two-Phase I/O Optimization



Two-Phase Read Algorithm

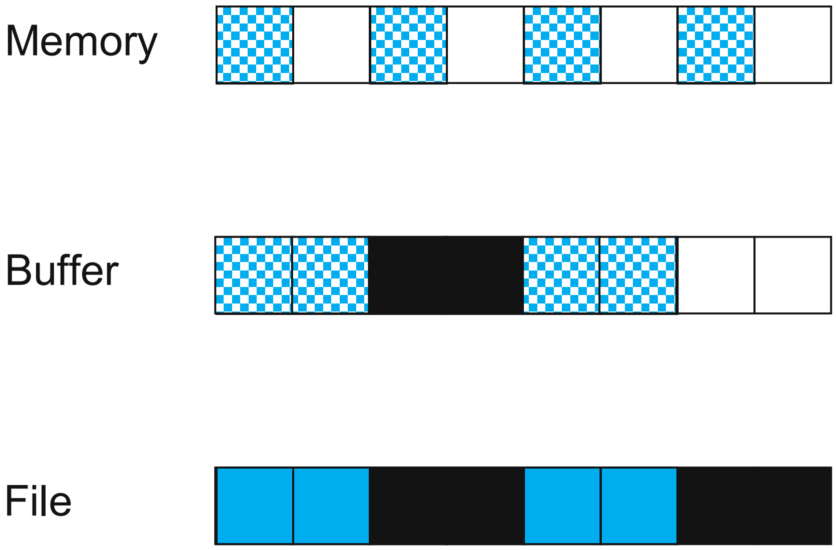
- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second “phase” redistributes data to final destinations
- Two-phase writes operate in reverse (redistribute then I/O)
  - Typically read/modify/write (like data sieving)
  - Overhead is lower than independent access because there is little or no false sharing
- Aggregating to fewer nodes as part of this process is trivial (and implemented!)

Challenge: Noncontiguous I/O



- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

Noncontiguous I/O: Data Sieving



Data Sieving Read Sequence

- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Similar to how a block-based file system interacts with storage
- Generally very effective, but not as good as having a PFS that supports noncontiguous access

MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations that result in better I/O performance

- Still a big gap between application and MPI-IO storage models
- Forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

Higher Level I/O Interfaces

Challenge: Improving Usability of Storage

- High level libraries are designed to make life easier for application writers
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays
- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Both of our example interfaces are implemented on top of MPI-IO

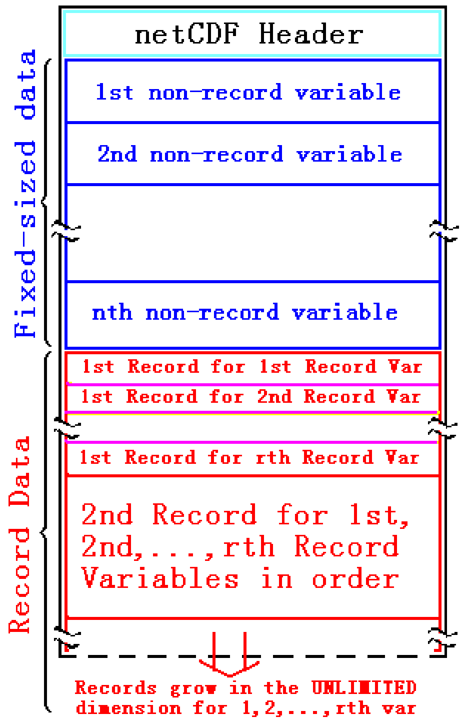
PnetCDF Interface and File Format

Parallel netCDF (PnetCDF)

- Based on original "Network Common Data Format" (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C and Fortran interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
- Unrelated to netCDF-4 work

netCDF/PnetCDF Files

- PnetCDF files consist of three regions
  - Header
  - Non-record variables (all dimensions specified)
  - Record variables (ones with an unlimited dimension)
- Record variables are interleaved, so using more than one in a file is likely to result in poor performance due to noncontiguous accesses
- Data is always written in a big-endian format



Data in PnetCDF

- Write case: “bimodal”
- Create a dataset (file)
  - Puts dataset in define mode
  - Allows us to describe the contents
    - *Define dimensions for variables*
    - *Define variables using dimensions*
    - *Store attributes if desired (for variable or dataset)*
- Switch from define mode to data mode to write variables
- Store variable data
- Close the dataset
- Read case similar:
  - No define mode
  - Query dataset for attributes, variables
  - Read data

Example: FLASH with PnetCDF

- FLASH AMR structures do not map directly to netCDF multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in netCDF multidimensional arrays
- Chose to
  - Place all checkpoint data in a single file
  - Impose a linear ordering on the AMR blocks
    - *Use 4D variables*
  - Store each FLASH variable in its own netCDF variable
    - *Skip ghost cells*
  - Record attributes describing run time, total blocks, etc.

Defining Dimensions

```
int status, ncid, dim_tot_blks, dim_nxb,
    dim_nyb, dim_nzb;
MPI_Info hints;
/* create dataset (file) */
status = ncmpi_create(MPI_COMM_WORLD, filename,
    NC_CLOBBER, hints, &file_id);
/* define dimensions */
status = ncmpi_def_dim(ncid, "dim_tot_blks",
    tot_blks, &dim_tot_blks);
status = ncmpi_def_dim(ncid, "dim_nxb",
    nzones_block[0], &dim_nxb);
status = ncmpi_def_dim(ncid, "dim_nyb",
    nzones_block[1], &dim_nyb);
status = ncmpi_def_dim(ncid, "dim_nzb",
    nzones_block[2], &dim_nzb);
```

Each dimension gets a unique reference

Creating Variables

```
int dims = 4, dimids[4];
int varids[NVARS];
/* define variables (X changes most quickly) */
dimids[0] = dim_tot_blks;
dimids[1] = dim_nzb;
dimids[2] = dim_nyb;
dimids[3] = dim_nxb;
for (i=0; i < NVARS; i++) {
     status = ncmpi_def_var(ncid, unk_label[i],
        NC_DOUBLE, dims, dimids, &varids[i]);
}
```

Same dimensions used for all variables

Writing Variables

```
double *unknowns; /* unknowns[blk][nzb][nyb][nxb] */
size_t start_4d[4], count_4d[4];
start_4d[0] = global_offset; /* different for each process */
start_4d[1] = start_4d[2] = start_4d[3] = 0;
count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;  count_4d[3] = nxb;
for (i=0; i < NVARS; i++) {
      /* ... build datatype “mpi_type” describing values of a single variable ...
         */
      /* collectively write out all values of a single variable */
      ncmpi_put_vara_all(ncid, varids[i], start_4d, count_4d,
            unknowns, 1, mpi_type);
}
status = ncmpi_close(file_id);
```

Typical MPI buffer-count-type tuple

Inside PnetCDF Define Mode

- In define mode (collective)
  - Use MPI_File_open to create file at create time
  - Set hints as appropriate (more later)
  - Locally cache header information in memory
    - *All changes are made to local copies at each process*
- At ncmpi_enddef
  - Process 0 writes header with MPI_File_write_at
  - MPI_Bcast result to others
  - Everyone has header data in memory, understands placement of all variables
    - *No need for any additional header I/O during data mode!*

Inside PnetCDF Data Mode

- Inside ncmpi_put_vara_all (once per variable)
  - Each process performs data conversion into internal buffer
  - Uses MPI_File_set_view to define file region
    - *Contiguous region for each process in FLASH case*
  - MPI_File_write_all collectively writes data
- At ncmpi_close
  - MPI_File_close ensures data is written to storage

- MPI-IO performs optimizations
  - Two-phase possibly applied when writing variables
- MPI-IO makes PFS calls
  - PFS client code communicates with servers and stores data

PnetCDF Wrap-Up

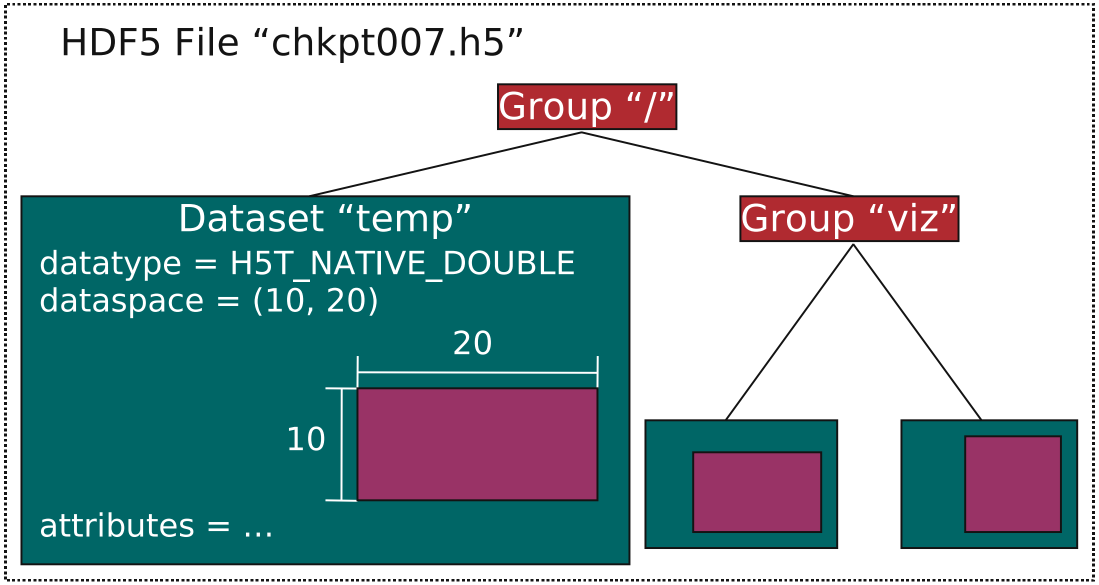
- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - One record's worth of record variable: < 4 GiB
  - 2^{32} -1 records

HDF5 Interface and File Format

HDF5

- Hierarchical Data Format, from the HDF Group (formerly of NCSA)
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on dataset, data
- Features:
  - C, C++, and Fortran interfaces
  - Portable data format
  - Optional compression (not in parallel I/O mode)
  - Data reordering (chunking)
  - Noncontiguous I/O (memory and file) with hyperslabs

HDF5 Files



- HDF5 files consist of groups, datasets, and attributes
  - Groups are like directories, holding other groups and datasets
  - Datasets hold an array of typed data
    - *A datatype describes the type (not an MPI datatype)*
    - *A dataspace gives the dimensions of the array*
  - Attributes are small datasets associated with the file, a group, or another dataset
    - *Also have a datatype and dataspace*
    - *May only be accessed as a unit*

HDF5 Data Chunking

- Apps often read subsets of arrays (subarrays)
- Performance of subarray access depends in part on how data is laid out in the file
  - e.g. column vs. row major
- Apps also sometimes store sparse data sets
- Chunking describes a reordering of array data
  - Subarray placement in file determined lazily
  - Can reduce worst-case performance for subarray access
  - Can lead to efficient storage of sparse data
- Dynamic placement of chunks in file requires coordination
  - Coordination imposes overhead and can impact performance

Inside HDF5

- MPI_File_open used to open file
- Because there is no "define" mode, file layout is determined at write time
- In HDF write call:
  - Processes communicate to determine file layout
    - *Process 0 performs metadata updates*
  - Call MPI_File_set_view
  - Call MPI_File_write_all to collectively write
    - *If this was turned on*
- User could have defined noncontiguous region in memory or file
- In FLASH application, data is kept in native format and converted at read time (defers overhead)
  - Could store in some other format if desired
- At the MPI-IO layer:
  - Metadata updates at every write are a bit of a bottleneck
    - *MPI-IO from process 0 introduces some skew*

Concluding Remarks

Wrapping Up

- Computational science applications present a complex set of challenges with respect to their I/O needs
  - Very high degrees of concurrency in access
  - Very high bandwidth requirements, bursty I/O
  - Effective means for mapping scientific data models into storage structures
- A layered software architecture has evolved (and is still evolving) to address the needs of these applications
  - Relies on adequate hardware resources
  - Also typically relies on a commercial parallel file system
  - Software specific to HPC helps bridge the gap
- The gap is growing between the needs of computational science applications and the capabilities offered by storage vendors and commercial parallel file systems
  - Opportunities for new approaches to make their way into the I/O software stack

Printed References

- John May, Parallel I/O for High Performance Computing, Morgan Kaufmann, October 9, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF
- William Gropp, Ewing Lusk, and Rajeev Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, November 26, 1999.
  - In-depth coverage of MPI-IO API, including a very detailed description of the MPI-IO consistency semantics

On-Line References (1 of 3)

- netCDF and netCDF-4
  - http://www.unidata.ucar.edu/packages/netcdf/
- PnetCDF
  - http://www.mcs.anl.gov/parallel-netcdf/
- ROMIO MPI-IO
  - http://www.mcs.anl.gov/romio/
- HDF5 and HDF5 Tutorial
  - http://www.hdfgroup.org/
  - http://hdf.ncsa.uiuc.edu/HDF5/
  - http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html

On-Line References (2 of 3)

- PVFS
  http://www.pvfs.org/
- Lustre
  http://www.lustre.org/
- GPFS
  http://www.almaden.ibm.com/storagesystems/file_systems/GPFS/

On-Line References (3 of 3)

- LLNL I/O tests (IOR, fdtree, mdtest)
  - http://www.llnl.gov/icc/lc/siop/downloads/download.html
- Parallel I/O Benchmarking Consortium (noncontig, mpi-tile-io, mpi-md-test)
  - http://www.mcs.anl.gov/pio-benchmark/
- FLASH I/O benchmark
  - http://www.mcs.anl.gov/pio-benchmark/
  - http://flash.uchicago.edu/~jbgallag/io_bench/ (original version)
- b_eff_io test
  - http://www.hlrs.de/organization/par/services/models/mpi/b_eff_io/
- mpiBLAST
  - http://www.mpiblast.org

Acknowledgements

This work is supported in part by U.S. Department of Energy Grant DE-FC02-01ER25506, by National Science Foundation Grants EIA-9986052, CCR-0204429, and CCR-0311542, and by the U.S. Department of Energy under Contract W-31-109-ENG-38.

This work was performed under the auspices of the U.S. Department of Energy by LLNS, LLC, Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.